ПРОГРАММИРОВАНИЕ, 2020, № 4, с. 14–26

ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ

УДК 004.421.6

ДЕДУКТИВНАЯ ВЕРИФИКАЦИЯ REFLEX-ПРОГРАММ

© 2020 г. И. С. Ануреев[a,b,*], Н. О. Гаранина[a,b,], Т. В. Лях[b,***], А. С. Розов[b,****], В. Е. Зюбин[b,*****], С. П. Горлач[c,******]**

[a] *Институт систем информатики им. А.П. Ершова СО РАН*
630090 Новосибирск, пр. ак. Лаврентьева, д. 6, Россия

[b] *Институт автоматики и электрометрии СО РАН*
630090 Новосибирск, проспект Академика Коптюга, д. 1, Россия

[c] *Университет Мюнстера*
48149 Мюнстер, ул. Эйнштейна, д. 62, Германия

**E-mail: anureev@gmail.com*
***E-mail: garanina@iis.nsk.su*
****E-mail: antsys_nsu@mail.ru*
*****E-mail: rozov@iae.nsk.su*
******E-mail: zyubin@iae.nsk.su*
*******E-mail: gorlatch@uni-muenster.de*

Поступила в редакцию 10.02.2020 г.
После доработки 20.02.2020 г.
Принята к публикации 15.03.2020 г.

В этой статье представлен новый двухшаговый метод верификации для управляющего программного обеспечения. Новизна метода состоит в том, чтобы свести проверку временных свойств управляющей программы к дедуктивной верификации императивной программы в стиле Хоара, которая явно моделирует время и историю выполнения управляющей программы. Метод применяется к программам на языке Reflex – предметно-ориентированном расширении языка C, разработанном в качестве альтернативы языкам стандарта IEC 61131-3. Reflex – это процесс-ориентированный язык, описывающий управляющие программы в терминах взаимодействующих процессов, управляемых событиями операций и операций с дискретными временными интервалами. На первом шаге аннотированная Reflex-программа транслируется в эквивалентную аннотированную императивную программу на ограниченном подмножестве языка C, расширенном логическим типом bool и супертипом value, объединяющем значения, которые могут возвращать функции и операции языка Reflex, а также оператором havoc x, присваивающем произвольное значение переменной x. На втором шаге выполняется дедуктивная верификация полученной императивной программы. Мы иллюстрируем наш метод на примере дедуктивной верификации Reflex-программы, управляющей сушилкой для рук. Пример включает исходную Reflex-программу, набор требований, результирующую аннотированную программу, порожденные условия корректности и результаты доказательства этих условий в Z3py – интерфейсе к SMT-решателю Z3, представленном на языке Python.

DOI: 10.31857/S0132347420040020

1. ВВЕДЕНИЕ

Растущая сложность систем управления, используемых в нашей повседневной жизни, требует переоценки инструментов проектирования и разработки. Такая переоценка особенно важна для критических с точки зрения безопасности систем, где неправильное поведение и/или отсутствие надежности может привести к неприемлемой потере средств или даже человеческой жизни. Такие системы широко распространены в промышленности, особенно на химических и металлургических заводах. Поскольку поведение сложных систем управления определяется программным обеспечением, исследование управляющего программного обеспечения представляет большой интерес. Корректное поведение в различных условиях окружающей среды должно быть гарантировано. При отказе оборудования, например, поломках механических частей объекта управления или исполнительных устройств, система должна автоматически сформировать управляющие воздействия, которые предотвращают опасные последствия таких отказов. Общепринятое название этого свойства системы управления – отказоустойчивость (fault tolerant behavior) [1].

Из-за специфики предметной области системы управления обычно основаны на промышленных контроллерах, также известных как программируемые логические контроллеры (ПЛК), и специализированные языки используются для проектирования управляющего программного обеспечения для таких систем. Наиболее распространенными в области программирования ПЛК являются языки стандарта IEC 61131-3 [2]. Однако, поскольку сложность управляющего программного обеспечения возрастает, а качество становится более приоритетным, 35-летняя технология, основанная на подходе IEC 61131-3, не всегда способна удовлетворить современные требования [3]. Поэтому исследователи предпринимают попытки либо модифицировать модель разработки IEC 61131-3, например, обогатить модель объектно-ориентированными концепциями [4], либо вообще предложить взамен IEC 61131-3 альтернативные понятийно-языковые средства, например, [5–8].

Для устранения ограничений и проблем при разработке современного комплексного управляющего программного обеспечения в [9] была предложена парадигма процесс-ориентированного программирования. Эта парадигма представляет управляющее программное обеспечение в виде набора взаимодействующих процессов, где процессы – это конечные автоматы, дополненные неактивными состояниями и специальными функциями управления процессами и обработки временных интервалов. По сравнению с известными модификациями конечных автоматов, например, последовательными взаимодействующими процессами [10], диаграммами состояний Харела [11], автоматами ввода/вывода [12], Esterel [13], гибридными автоматами [14], исчислением взаимодействующих систем [15] и их временными расширениями [16, 17], этот метод обеспечивает средства для спецификации параллелизма и при этом сохраняет линейность потока управления на уровне процесса. Таким образом, он обеспечивает концептуальную основу для разработки процессно-ориентированных языков, предназначенных для создания программного обеспечения для ПЛК. Процесс-ориентированный подход был реализован в таких предметно-ориентированных языках как SPARM [18], Reflex [19] и IndustrialC [20]. Эти языки являются С-подобными и, поэтому, просты в изучении. Трансляторы для них порождают С-код, который обеспечивает кросс-платформенную переносимость. Благодаря прямой поддержке конечных автоматов и операций с плавающей запятой, эти языки позволяют достаточно просто создавать программное обеспечение для ПЛК.

Язык SPARM является предшественником языка Reflex и в настоящее время не используется. IndustrialC нацелен на использование периферийных устройств микроконтроллера (регистров, таймеров, ШИМ и т.д.) и расширяет Reflex средствами обработки прерываний. Программа на языке Reflex специфицируется как набор взаимодействующих параллельных процессов. Reflex включает специализированные инструкции для управления процессами и обработки временных интервалов. Он также предоставляет инструкции для связывания программных переменных с физическими сигналами ввода/вывода. Процедуры чтения/записи данных через регистры и их сопоставление с переменными порождаются транслятором автоматически.

Reflex предполагает цикл управления с фиксированным временем исполнения и строгую инкапсуляцию зависимых от платформы подпрограмм ввода-вывода в библиотеку, что является широко применяемым методом в системах на основе IEC 6113-3. Для обеспечения простоты поддержки и межплатформенной переносимости генерация исполняемого кода осуществляется в два этапа: транслятор Reflex генерирует С-код, а затем С-компилятор создает исполняемый код для целевой платформы. Reflex – простой язык с точки зрения операторов и типов данных. Он не имеет указателей, массивов и циклов. Несмотря на простоту, этот язык был успешно применен в нескольких индустриальных системах управления, например, в программном обеспечении для управления печью установкой для выращивания монокристаллического кремния [21]. Семантическая простота языка вместе с практической применимостью делает Reflex привлекательным для теоретических исследований.

В настоящее время проект Reflex сфокусирован на средствах проектирования и разработки критических с точки зрения безопасности систем. Благодаря своей системной независимости Reflex легко интегрируется со сторонними средствами, например, с LabVIEW [22]. Это позволяет разрабатывать программное обеспечение, сочетающее управляемое событиями поведение с расширенным графическим интерфейсом пользователя, удаленными датчиками и исполнительными устройствами, устройствами с поддержкой LabVIEW и т.д. Используя гибкость LabVIEW, был разработан набор тренажеров для учебных целей [23]. Основанные на LabVIEW симуляторы включают в себя 2D-анимацию, инструменты для отладки и языковую поддержку для обучения разработке управляющего программного обеспечения. Одним из результатов, полученных в этом направлении, является набор инструментов динамической верификации на основе LabVIEW для Reflex-программ. Динамическая верификация рассматривает программное обеспечение как черный ящик и проверяет его соответствие требованиям, наблюдая за поведением программного обеспечения во время выполнения на наборе тестовых случаев. Хотя такая процедура может помочь обнаружить наличие ошибок в программном обеспечении, она не может гарантировать их отсутствие [24].

В отличие от динамической проверки, формальные методы обычно признаются как единственный способ обеспечить полноту верификации проверяемого свойства программного обеспечения. Поэтому очень важно адаптировать формальные методы верификации для Reflex-программ.

В этой статье мы предлагаем метод дедуктивной верификации Reflex-программ. Оригинальная двухшаговая схема метода позволяет нам свести проверку временных свойств Reflex-программ к дедуктивной верификации C-подобной программы, которая явно моделирует время и историю выполнения Reflex-программы.

Работа имеет следующую структуру. В разделе 2, мы формулируем подход к спецификации временных свойств Reflex-программ и пример Reflex-программы, управляющей сушилкой для рук, а также ее временные свойства. В разделе 3 на примере программы управления сушилкой для рук описывается алгоритм трансляции аннотированных Reflex-программ в язык VOL[Reflex] (Verification-Oriented Language) – ограниченное подмножество аннотированных императивных программ на языке C, расширенное логическим типом bool и супертипом value, объединяющем значения, которые могут возвращать функции и операции языка Reflex, а также оператором havoc x, присваивающем произвольное значение переменной x. В разделе 4 представлена формальная трансляционная семантика языка Reflex в язык VOL[Reflex] (далее VOL), описывающая правила трансляции Reflex-инструкций в VOL-инструкции. Трансляция аннотированной Reflex-программы в VOL-программу – первый шаг нашего метода дедуктивной верификации. Второй шаг – дедуктивная верификация VOL-программы, состоящая в порождении условий корректности для нее и их доказательстве, – определяется в разделе 5. Он включает алгоритм порождения условий корректности и примеры условий корректности для VOL-программы – результата трансляции программы управления сушилкой для рук. В заключительном разделе 6 мы обсуждаем особенности нашего метода и направления будущих исследований.

2. СПЕЦИФИКАЦИЯ ВРЕМЕННЫХ СВОЙСТВ REFLEX-ПРОГРАММ

Наш метод верификации сводит верификацию Reflex-программ к верификации VOL-программ. Reflex-программа вместе с ее проверяемыми временными свойствами транслируется в эквивалентную VOL-программу с соответствующим набором проверяемых свойств. В этом разделе мы определим подход к спецификации для свойств Reflex-программ. Этот подход иилюстрируется на примере Reflex-программы, управляющей сушилкой для рук.

Мы специфицируем свойства Reflex-программ, используя два языка: язык аннотаций и язык аннотаторов. *Язык аннотаций* – это язык логических формул – *аннотаций*, описывающих свойства программы. *Язык аннотаторов* – это язык разметки, сопоставляющий аннотации конструкциям программы. Элементы этого языка называются *аннотаторами*. Программа, расширенная аннотаторами, называется *аннотированной программой*.

Аннотации Reflex-программ являются формулами многосортной логики первого порядка, представленными в синтаксисе интерфейса Z3py [25] к SMT-решателю Z3 [26], который используется для доказательства условий корректности, порождаемых для VOL-программ – результатов трансляции Reflex-программ.

Временные свойства Reflex-программ могут выражаться в этих аннотациях за счет использования модели дискретного времени, глобального таймера, локальных таймеров процессов и историй значений программных переменных.

Модель дискретного времени основана на периодичности взаимодействия между Reflex-программой и объектом управления. Reflex-программа и управляемый ею объект взаимодействуют через входные и выходные порты, связанные с программными переменными. В начале каждого цикла управления программа читает значения из входных портов и записывает их в соответствующие переменные. Изменение значения переменной в результате записи значения из входного порта называется *внешней модификацией* переменной. В конце цикла управления программа записывает новые значения в выходные порты. Запись значений из входных портов в переменные и чтение значений из

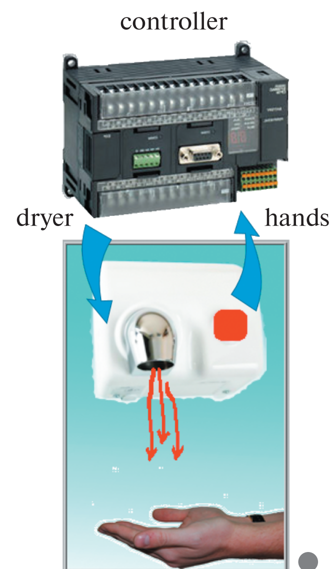

Рис. 1. Hand Dryer.

переменных в выходные порты происходит периодически с фиксированным периодом (программным циклом), измеряемым в миллисекундах. Выполнение одного программного цикла называется тактом работы Reflex-программы.

Время в аннотациях моделируется глобальным таймером и локальными таймерами процессов. Таймеры задаются переменными типа `int`. Эти переменные не могут быть переменными Reflex-программы. *Глобальный таймер* `timer` подсчитывает число тактов работы Reflex-программы. Каждый программный процесс p_i имеет свой *локальный таймер* $p_{i_\mathtt{timer}}$, который также как и глобальный таймер считает время в количестве тактов работы программы.

Каждая программная переменная x интерпретируется в аннотациях как массив, в котором индекс i означает i-й цикл выполнения программы, а элемент $x[i]$ хранит значение переменной x на этом цикле. Таким образом, в контексте аннотации, x хранит историю своих изменений.

Аннотации также используют следующие переменные:

- `val` – для доступа к значению, возвращаемую функцией или операцией в Reflex-программе;
- `val_1`, ..., `val_`*n* – для хранения значений аргументов функций и операций, вычисляемых в Reflex-программе (n – максимальная местность функций и операций, используемых в программе);
- $p_{i_\mathtt{state}}$ – для доступа к текущему состоянию программного процесса p_i.

Обозначим множество аннотаций F, так что $f \in F$ является аннотацией, специфицирующей некоторое свойство Reflex-программы.

Язык аннотаторов для Reflex-программ включает четыре вида аннотаторов. *Инвариантный аннотатор* INV f; специфицирует, что свойство f должно выполняться перед каждым программным циклом. *Аннотатор начального условия* `ICON` f; специфицирует, что свойство f должно выполняться перед первым программным циклом. *Аннотатор внешнего условия* `ECON` f; накладывает ограничения на внешние модификации переменных: свойство f должно выполняться после каждой внешней модификации. *Аннотатор функций* `REQUIRES` P_f; `ENSURES` Q_f; должен размещаться непосредственно после прототипа функции $t\ f\ (t_1\ x_1, \ldots, t_n\ x_n)$. Прототипы функций используются чтобы вызывать функции, написанные на других языках программирования, в Reflex-программах. Этот аннотатор специфицирует предусловие P_f и постусловие Q_f функции f. Формулы P_f и Q_f зависят от $x_1, \ldots, x_n$. Постусловие Q_f также зависит от `val`, которая хранит значение, возвращаемое f.

Проиллюстрируем наш подход к спецификации временных свойств на примере программы управления сушилкой для рук (рис. 1, Листинг 1).

Программа использует вход с инфракрасного датчика, показывающего присутствие или отсутствие рук под сушилкой, и, на основании этих данных, управляет вентилятором и обогревателем через общий выходной сигнал.

Сформулируем несколько временных свойств для этой программы. Первое свойство заключается в том, что сушилка должна включиться за время, приемлемое с точки зрения пользователя (не позднее 100 миллисекунд), когда под ней появляются руки. Второе свойство постулирует, что сушилка никогда не включается произвольно. Эти свойства задаются ниже формулами r_1 и r_2.

```
PROGR HandDryerController {
 /* =========================== */
 /* == ANNOTATIONS : */
 /* INV inv; */
 /* ICON True ; */
 /* ECON True ; */
 /* == END OF ANNOTATIONS */
 TACT 100;
 CONST ON TRUE ;
 CONST OFF FALSE ;
 /* ============================= */
 /* I/O ports specification */
 /* direction , name , address , */
 /* offset , size of the port */
 /* ============================= */
 INPUT SENSOR_PORT 0 0 8;
 OUTPUT ACTUATOR_PORT 1 0 8;
```

```
/* ============================== */
/* processes definition */
/* ============================== */
PROC Ctrl {
 /* ===== VARIABLES ============= */
 BOOL hands = { SENSOR_PORT [1]} FOR ALL;
 BOOL dryer = { ACTUATOR_PORT [1]} FOR ALL;
 /* ===== STATES ================ */
 STATE Waiting {
  IF ( hands == ON) {
   dryer = ON;
   SET NEXT ;
  } ELSE dryer = OFF;
 }
 STATE Drying {
  IF ( hands == ON) RESET TIMEOUT ;
  TIMEOUT 10 SET STATE Waiting ;
 }
} /* \ PROC */
} /* \ PROGRAM */
```

Листинг 1: Reflex-программа управления сушилкой для рук

В Reflex-программах инструкция PROGR специфицирует имя и тело программы. Аннотаторы добавляются в начало тела программы как специальный вид комментариев. В нашем случае аннотаторами являются `INV inv;`, `ICON True;` и `ECON True;` (аннотация `inv` определена ниже). Инструкция `TACT` специфицирует число миллисекунд, соответствующее одному программному циклу. Инструкция `CONST` используется для объявления программных констант. Инструкции `INPUT` и `OUTPUT` описывают входные и выходные порты, соответственно. Программные переменные определяются через объявления переменных. Например, объявление переменной `BOOL hands = SENSOR_PORT[1] FOR ALL` связывает логическую переменную `hands` с первым битом порта `SENSOR_PORT` и указывает, что все процессы могут использовать эту переменную. Инструкция `PROC` используется для описания процессов программы. В нашем примере программа имеет один процесс `Ctrl` (контроллер), который управляет сушилкой для рук, то есть вентилятором и обогревателем. Инструкция `STATE` объявляет возможные состояния процесса. Процесс `Ctrl` может находиться в двух состояниях `WAITING` и `DRYING`. Действия, выполняемые процессом в состоянии, описываются в теле этого состояния с помощью операторов и операций.

В дополнении к C-подобным операторам и операциям, язык Reflex включает ряд специализированных операторов и операций для работы с процессами. Оператор `RESET TIMEOUT;` сбрасывает локальный таймер текущего процесса. Оператор `TIMEOUT` x stm; запускает выполнение оператора stm, когда локальный таймер текущего процесса достигает значения x. Оператор `SET NEXT;` переключает текущий процесс в следующее состояние (в порядке, в котором состояния объявляются в теле процесса), а оператор `SET STATE s;` переключает процесс в состояние s. Эти два оператора также обнуляют локальный таймер процесса.

Программа управления сушилкой не накладывает ограничения на начальное и внешнее условия. Они определяются как True. Заметим, что если бы язык проекций не включал тип `bool` как в [27] и константы ON и OFF определялись бы как `1` и `0`, то эти условия имели бы вид:

$$And(Or(\texttt{dryer[0]==0}, \texttt{dryer[0]==1}), Or(\texttt{hands[0]==0}, \texttt{hands[0]==1})$$

и

$$Or(\texttt{hands[timer]==0}, \texttt{hands[timer]==1}),$$

специфицируя, что переменные `dryer` и `hands` могут принимать только значения `0` и `1`. Включение типа `bool` позволяет упростить аннотации и, соответственно, порождаемые условия корректности.

Инвариант `inv` вида $And(r_1, r_2, ap)$ включает свойства r_1 и r_2, которые специфицируют желаемое поведение программы и конъюнкцию ap вспомогательных свойств, необходимых для их верификации. Эти вспомогательные свойства формулируются следующим образом: 1) значения программных констант равны их преодопределенным значениям, 2) глобальный таймер и все локальные таймеры неотрицательны, 3) текущие значения перемен-

ных hands и dryer совпадают с их предыдущими значениями (это свойство обеспечивает переход на новый такт выполнения программы), 4) сушилка может находиться только в двух состояниях WAITING и DRYING, и 5) сушилка в состоянии DRYING всегда включена. Мы опускаем формальную спецификацию для *ap*, так как она довольно громоздка.

Свойство r_1 вида

```
ForAll(i,
   Implies(
     And(0 <= i, i < timer),
     Implies(
        And(
          Implies(i > 0, hands[i - 1]==OFF),
          hands[i]==ON),
        dryer[i]==ON)))
```

специфицирует, что сушилка включается (dryer[i]==ON) не позднее, чем через 100 миллисекунд (один такт) после появления рук.

Свойство r_2 вида

```
ForAll(i,
  Implies(
    And(0 <= i, i < timer - 1),
    Implies(
      And(dryer[i]==OFF,
            hands[i + 1]==OFF),
      dryer[i + 1]==OFF)))
```

описывает требование, что сушилка никогда не включается самопроизвольно.

В следующем разделе мы представим на примере программы управления сушилкой для рук алгоритм трансляции аннотированной программы Reflex в VOL-программу, для которой порождаются условия корректности.

3. ТРАНСЛЯЦИЯ АННОТИРОВАННОЙ REFLEX-ПРОГРАММЫ В VOL-ПРОГРАММУ

Reflex-программы и VOL-программы используют один и тот же язык аннотаций. Язык аннотирования для VOL-программ включает аннотатор функций и три дополнительных аннотатора. Аннотатор ASSUME *f*; специфицирует, что *f* считается истинной в точке размещения этого аннотатора в программе. Аннотатор ASSERT *f*; утверждает, что *f* должна быть истинной в точке размещения этого аннотатора в программе. Инвариантный аннотатор INV *l f*; – специальный вариант именованного ASSERT-аннотатора с именем *l*, который обрабатывается специальным образом нашим алгоритмом порождения условий корректности.

VOL-программа – результат трансляции Reflex-программы управления сушилкой для рук – имеет следующий вид:

```
# define TACT 100
# define ON TRUE
# define OFF FALSE
# define STOP_STATE 0
# define ERROR_STATE 1
# define Ctrl_Waiting 2
# define Ctrl_Drying 3

value val;
int timer ;
int Ctrl_state ;
int Ctrl_timer ;
bool hands [];
bool dryer [];

inline void init () {
 timer = 0;
 Ctrl_state = Ctrl_Waiting ;
 Ctrl_timer = 0;
 ASSUME True ;
}

inline void Ctrl_exec () {
 switch ( Ctrl_state ) {
  case Ctrl_Waiting :
   if ( hands [ timer ] == ON) {
   dryer [ timer ] = ON;
   Ctrl_timer = 0;
   Ctrl_state = Ctrl_Drying ;
  }
else
  dryer [ timer ] = OFF;
  break ;
case Ctrl_Drying :
  if ( hands [ timer ] == ON) {
   Ctrl_timer = 0;
   Ctrl_state = Ctrl_Drying ;
  }
  if ( Ctrl_timer >= 10) {
   Ctrl_timer = 0;
   Ctrl_state = Ctrl_Waiting ;
  }
  break ;
 }
}

void main () {
```

```
  init ();
  for (;;) {
   INV lab inv;
   havoc hands [ timer ];
   ASSUME True ;
   Ctrl_exec ();
   Ctrl_timer = Ctrl_timer + 1;
   timer = timer + 1;
   hands [ timer ] = hands [timer -1];
   dryer [ timer ] = dryer [timer -1];
  }
}
```

Листинг 2: VOL-программа управления сушилкой для рук

Рассмотрим отношения между инструкциями исходной Reflex-программы и результирующей VOL-программы.

Макроконстанта TACT, специфицирующая время программного цикла, заменяет конструкцию TACT. Константы Reflex-программы (например, ON и OFF) также заменяются макроконстантами. Макроконстанты STOP_STATE и ERROR_STATE кодируют состояние останова (специфицирующее, что выполнение процесса на текущем такте завершается нормально) и состояние ошибки (специфицирующее, что выполнение процесса на текущем такте завершается с ошибкой). Для каждого программного процесса p_i и для каждого состояния s этого процесса, макроконстанта p_i_s кодирует это состояние.

Переменные val, $\texttt{val}_1$, ..., val_n рассматриваются как глобальные переменные VOL-программы типа value. В нашем случае, $n = 0$, так в программе не вычисляются функции и операции. Переменная timer специфицирует глобальный таймер. Для каждого программного процесса p_i, переменные p_i_state и p_i_timer специфицируют текущее состояние и локальный таймер процесса p_i. Тип t каждой Reflex-переменной x заменяется типом динамического массива t [].

Функция init() инициализирует процессы программы. Она устанавливает значение 0 для глобального таймера и всех локальных таймеров, устанавливает первый процесс в его начальное состояние и все остальные процессы в состояние останова и накладывает ограничения на начальные значения Reflex-переменных, используя аннотатор ASSUME f (для программы управления сушилкой для рук ASSUME *True*).

Для каждого процесса p_i, функция p_i_exec специфицирует действия процесса p_i во время программного цикла. Тело функции p_exec представляет оператор switch, где метки – макроконстанты, кодирующие состояния процесса p_i_exec. Все специфические для языка Reflex операторы и операции в телах состояний процессов заменяются C-инструкциями в соответствие с их семантикой.

Бесконечный цикл for(;;), специфицирующий действия всех процессов во время программного цикла является последним оператором результирующей программы. Его тело начинается с инвариантного аннотатора INV *lab inv*;, специфицирующего инвариант *inv* Reflex-программы. Следующий фрагмент

havoc hands[timer]; ASSUME *True*;

специфицирует внешние модификации Reflex-переменных (в нашем случае, hands) и ограничение *True* на них. Оператор havoc x;, добавленный в VOL из [28], моделирует присваивание произвольного значения переменной x из множества значений типа этой переменной. Третий фрагмент – это последовательность вызовов функций p_i_exec() для каждого процесса p_i. Следующий фрагмент увеличивает на 1 значения глобального таймера и всех локальных таймеров. Последний фрагмент специфицирует, что значения Reflex-переменных сохраняются после инкрементации глобального таймера, что соответствует переходу к новому такту выполнения Reflex-программы. Для программы управления сушилкой для рук, этот фрагмент имеет вид:

hands[timer] = hands[timer-1];
dryer[timer] = dryer[timer-1];.

В следующем разделе мы опишем общий вид правил трансляции Reflex-программ в VOL-программы (трансляционную семантику аннотированных Reflex-программ).

4. ТРАНСЛЯЦИОННАЯ СЕМАТИКА АННОТИРОВАННЫХ REFLEX-ПРОГРАММ

Пусть C_R и C_V – множество инструкций языков Reflex и VOL, соответственно. Программы на этих языках также включаются в эти множества. Пусть $C = C_R \cup C_V$. Трансляционная семантика языка Reflex задается бинарным отношением $\rightsquigarrow \in C \times C$ таким, что $\neg(c \rightsquigarrow c)$ для $c \in C_C$. Это отношение определяется правилами трансляции для Reflex-инструкций.

Программа. Для Reflex-программы p определим: $\{p_1, \ldots, p_n\}$ – множество имен процессов; dec_T – декларация такта; dec_c, dec_v и dec_f – последовательности деклараций констант, переменных и внешних функций; m_i – число состояний процесса p_i; $s_{i1}, \ldots, s_{im_i}$ – последовательность имен состояний p_i в порядке их определения в программе p; $body(s_{ik_i})$ – тело состояния s_{ik_i}, из которого исключены декларации переменных; $V = \{v_1, \ldots, v_k\}$ – множество имен переменных программы p; $V_e = \{v_{e1}, \ldots, v_{es}\} \subseteq V$ – подмножество имен переменных, изменяемых объектом управления. Закоди-

руем номера состояний процессов последовательными натуральными числами, начиная с 2 (коды 0 и 1 используются для состояний останова и ошибки). Пусть $num(p_i, s_{ik_i})$ – код состояния s_{ik_i} процесса p_i. Тогда правило трансляции для p имеет вид:

```
p ⇝
decT decc
#define STOP_STATE 0
#define ERROR_STATE 1
#define p1_s11  num(p1,s11)
… // повторить для других состояний p1
… // повторить для других процессов
value val; value val_1; value val_ n;
int timer;
int p1_state; int p1_timer
… // повторить для других процессов
decv  decf
inline void init() {
  timer = 0;
  p1_state = p1_s11; p1_timer =0;
  p2_state = STOP_STATE; p2_timer =0;
  … // повторить для других процессов
  ASSUME icon;
}
dec1 … decn
void main() {
  init();
  for (;;) {
    INV lab inv;
    havoc ve1[timer];
    … // повторить для других переменных
    ASSUME econ;
    p1_exec(); …; pn_exec();
    p1_timer = p1_timer + 1;
    … // повторить для других процессов
    timer = timer + 1;
    v1[timer] = v1[timer-1];
    … // повторить для других переменных
 }
}
```

Здесь *icon* и *econ* – начальное и внешнее условия, соответственно; *inv* – инвариант программного цикла.

Определение dec_i функции p_i_exec, выполняющей процесс p_i в его текущем состоянии, имеет вид:

```
inline void pi_exec {
  switch (pi_state) {
    case pi_si1: body(si1) break;
    …
    case pi_simi: body(simi) break;
  }
}
```

Заметим, что мы не определяем семантику портов, а свойства программы задаем в терминах ее переменных. Поэтому декларации для портов отсутствуют в правой части правила для программы.

Правило подстановки. Правило подстановки позволяет применять правила трансляции к частям инструкций языка Reflex. Пусть c, c', c_1, $c_2 \in C$, и $c[c']$ обозначает место вхождения c' в c. Тогда правило подстановки имеет вид:

Если $c_1 \rightsquigarrow c_2$, то $c[c_1] \rightsquigarrow c[c_2]$.

Типы. Поскольку типы языка Reflex совпадают с типами языка VOL, правило трансляции для типов не требуется.

Декларация такта. Пусть k — число. Декларация такта определяется правилом:

TACT k; $\rightsquigarrow$ #define TACT k.

Декларации констант. Пусть u — имя константы, e — Reflex-выражение, и $\varnothing$ — пустой результат трансляции (удаление транслируемой инструкции). Декларации констант определяются правилами:

const u k; $\rightsquigarrow$ #define u k

const u e; $\rightsquigarrow$ #define u (e)

enum {u_1, u_2, …} $\rightsquigarrow$

#define u_1 0

enum {u_2 = 1, …}

enum {u_1 = k, u_2, …} $\rightsquigarrow$

#define u_1 k

enum {u_2 = $k+1$, …}

enum {} $\varnothing$

Декларации переменных. Декларации переменных определяются правилами:

t u …; $\rightsquigarrow$ t u;

FROM PROC pj u; $\rightsquigarrow \varnothing$.

Мы не учитываем в дедуктивной верификации связи переменных с портами и уровни доступа переменных. Поэтому, декларации Reflex-переменных непосредственно транслируются в декларации VOL-переменных.

Далее мы считаем, что Reflex-конструкции, для которых определяются правила, встречаются в теле некоторого состояния процесса p_i.

Операции над состояниями. Трансляционная семантика операций над состояниями определяется правилами:

(PROC IN STATE ACTIVE) $\rightsquigarrow$

(PROC p_i IN STATE ACTIVE);

(PROC p_j IN STATE ACTIVE) $\rightsquigarrow$

((p_j_state != STOP_STATE) &&

(p_j_state != ERROR_STATE));

(PROC IN STATE INACTIVE) $\rightsquigarrow$

(PROC p_i IN STATE INACTIVE);

(PROC p_j IN STATE INACTIVE) $\rightsquigarrow$

((p_j_state == STOP_STATE) ||

(p_j_state == ERROR_STATE));

(PROC IN STATE STOP) $\rightsquigarrow$

(PROC p_i IN STATE STOP);

(PROC p_j IN STATE STOP) $\rightsquigarrow$

(p_j_state == STOP_STATE);

(PROC IN STATE ERROR) $\rightsquigarrow$

(PROC p_i IN STATE ERROR);

(PROC p_j IN STATE ERROR) $\rightsquigarrow$

(p_j_state == ERROR_STATE).

Операторы управления состояниями. Операторы управления состояниями включают операторы STOP, ERROR, START, RESTART, SET и NEXT.

Оператор STOP определяется правилами:

STOP; $\rightsquigarrow$ STOP PROC p_i;

STOP PROC p_j; $\rightsquigarrow$

{p_j_timer=0; p_j_state= STOP_STATE;}.

Оператор ERROR определяется правилом:

ERROR; $\rightsquigarrow$

{p_i_timer = 0; p_i_state = ERROR_STATE;}.

Оператор START определяется правилом:

START PROC p_j; $\rightsquigarrow$

{p_j_timer = 0; p_j_state = p_j_s_1;}.

Оператор RESTART определяется правилом:

RESTART $\rightsquigarrow$ START PROC p_i;.

Оператор SET определяется правилом:

SET STATE s_{ij}; $\rightsquigarrow$

{p_i_timer = 0; p_i_state = p_i_s_{ij};}.

Оператор NEXT определяется правилом:

Если p_i_state $= s_{ij}$, то

SET NEXT; $\rightsquigarrow$

{p_i_timer = 0; p_i_state = $s_{i(j+1)}$;}.

Операторы управления таймаутами. Операторы управления таймаутами включают операторы RESET и TIMEOUT.

Оператор RESET определяется правилом:

RESET TIMEOUT; $\rightsquigarrow$ p_i_timer = 0;.

Пусть st — Reflex-оператор. Оператор TIMEOUT определяется правилом:

TIMEOUT e st $\rightsquigarrow$ if (p_i_timer $=e$) st.

Доказательство корректности трансляции (эквивалентности Reflex-программы и VOL-программы) не рассматривается в этой работе. Эквивалентность означает функциональную эквивалентность этих программ, где входами обеих программ являются вектора внешних модификаций для каждой Reflex-переменной, а выходами — вектора значений для каждой Reflex-переменной, так же как текущие состояния процессов и значения глобального и локальных таймеров. Определение эквивалентности основывается на операционной семантике аннотированных Reflex-программ и VOL-программ.

Таким образом, мы сводим верификацию временных свойств Reflex-программ к дедуктивной ве-

рификации VOL-программ. Дальше мы опишем правила порождения условий корректности для VOL-программ, примеры порожденных условий корректности для программы управления сушилкой для рук и их доказательство в SMT-решателе Z3.

5. ГЕНЕРАЦИЯ УСЛОВИЙ КОРРЕКТНОСТИ ДЛЯ C-ПРОЕКЦИЙ REFLEX-ПРОГРАММ

Подобно многим другим системам дедуктивной верификации, таким как FramaC [29], Spark [30], KeY [31], Dafny [32], и т.д., наш алгоритм порождения условий корректности определяет предикатный трансформер. Мы используем Z3 чтобы доказать порожденные условия корректности. Специфика алгоритма заключается в том, что он применяется к программе, которая является бесконечным циклом и некоторые переменные этой программы изменяются внешним образом на каждой итерации этого цикла. Алгоритм $sp(A, P)$ рекурсивно вычисляет сильнейшее постусловие [33], выраженное в многосортной логике первого порядка, для программного фрагмента A и предусловия P. Его применение начинается со всей программы и предусловия *True*. Результат его работы – множество условий корректности, сохраненное в переменной *vcs*. Алгоритм использует специальные переменные *vars* и *reached*. Переменная *vars* хранит информацию о переменных и их типах как множество пар вида $x{:}t$, где x – переменная, а t – ее тип. Переменная *reached* хранит множество имен инвариантных аннотаторов, которые были пройдены алгоритмом. Она используется чтобы гарантировать завершение алгоритма. Начальные значения этих переменных – пустые множества.

Мы определяем алгоритм порождения условий корректности *sp* как упорядоченное множество равенств вида $sp(A, P) = [a_1; ...; a_n; e]$. Эта нотация означает, что действия $a_1, ..., a_n$ последовательно выполняются перед тем, как выражение e вычисляется. Каждое действие a_i вида $v \mathrel{+}= S$ добавляет элементы из множества S в множество v. Равенство $sp(A, P) = e$ является сокращением для $sp(A, P) = [e]$.

Мы используем следующую нотацию в определении алгоритма. Пусть $array(t)$ обозначает тип массивов с элементами типа t. Пусть выражение e имеет тип t, $\{x{:}t, y{:}array(t)\} \subseteq vars$, $\{z{:}t, \ v{:}t\} \cap \cap vars = \varnothing$ для каждого t, и e' – результат преобразования VOL-выражения e в Z3py-выражение. Функция $Store(a, i, v)$ является функцией модификации массива из языка Z3.

В силу простоты языка VOL, алгоритм *sp* имеет следующую компактную форму:

1. $sp(t\ f(t_1 x_1, \ldots, t_n x_n);, \ P) =$
   $[vars \mathrel{+}= \{x_1{:}t_1, \ldots, x_n{:}t_n, ret_f{:}t\}; P];$
2. $sp(t\ x;, P) = [vars \mathrel{+}= \{x{:}t\}; P];$
3. $sp(\texttt{\#define}\ ce;, P) =$
   $[vars \mathrel{+}= \{c : t\};\ And(P, c == e')];$
4. $sp(\texttt{havoc}\ y[i];, P) =$
   $[vars \mathrel{+}= \{z{:}t, v{:}t\};$
   $And(P(y \leftarrow z), y == Store(z, i, v))];$
5. $sp(\texttt{havoc}\ x;, P) =$
   $[vars \mathrel{+}= \{z{:}t\};\ And(P(x \leftarrow z), x == z)];$
6. $sp(x[i] = e;\ , P) =$
   $[vars \mathrel{+}= \{z{:}array(t)\};$
   $And(P(y \leftarrow z), y == Store(z, i, e'(y \leftarrow z)))];$
7. $sp(x = e;\ , P) =$
   $[vars \mathrel{+}= \{z{:}t\};$
   $And(P(x \leftarrow z), x == e'(x \leftarrow z))];$
8. $sp(\{B\}, P) = sp(B, P);$
9. $sp(\texttt{if}\ (e)\ B\ \texttt{else}\ C, P) =$
   $Or(sp(B, And(P, econv(e))),$
   $sp(C, And(P, Not(econv(e)))));$

10. $sp($`switch` $(e)\ l_1{:}B_1$ `break;` $\ldots l_n{:}B_n$ `break;` $, P) =$
$Or(sp(B_1, And(P, e' == l_1)),$
$\ldots,$
$sp(B_n, And(P, e' == l_n)),$
$And(P, e' != l_1, \ldots, e' != l_n));$
11. $sp($`for(;;)` $\{B\}, P)\ sp(B$ `for(;;)` $\{B\}, P);$
12. $sp(f(e_1, \ldots, e_n);, P) =$
$sp(e_1;$ `val_1 := val;` $\ldots e_n;$ `val_n := val;`
`ASSERT` $P_f[x_1 \leftarrow$ `val_1` $, \ldots, x_n \leftarrow$ `val_n`$];$
`havoc val;`
`ASSUME` $Q_f[x_1 \leftarrow$ `val_1` $, \ldots, x_n \leftarrow$ `val_n`$];, P);$
13. $sp(e_1 + e_2;, P) =$
$sp(e_1;$ `val_1 := val;` $\ldots e_2;$ `val_2 := val;`
`val := (val_1 + val_2);` $, P);$

Другие операции определяются аналогичным образом. Заметим, что, для простоты, мы не рассматриваем приведение типов и считаем, что операции над числами в идеальной арифметике и машинной арифметике совпадают. Более точное определение операционной и аксиоматической семантик для операций может быть найдено в [34, 35].

14. $sp($`ASSUME` $e;, P) = And(P, e);$
15. $sp($`ASSERT` $e;, P) =$
$[vcs \mathrel{+}= \{Implies(P, e)\}; And(P, e)];$
16. если $l \notin reached$, то $sp($`INV` $le;\ A, P) =$
$[reached \mathrel{+}= \{l\}; vcs \mathrel{+}= \{Implies(P, e)\}; sp(A, e)];$
17. если $l \in reached$, то $sp($`INV` $le;\ A, P) =$
$[vcs \mathrel{+}= \{Implies(P, e)\}; e];$
18. $sp(stA, P) = sp(A, sp(st, P)).$

Этот алгоритм завершается, так как каждый вложенный вызов *sp* выполняется на меньшем программном фрагменте во всех случаях кроме `for(;;)` (случай 11), и, согласно случаю 16, алгоритм может проходить инвариантный аннотатор в начале тела цикла `for(;;)` только один раз.

Вычисление условий корректности для пути аннотированной программы, управляющей сушилкой для рук (Листинг 2), начинающегося в точке `#define TACT 100` и заканчивающегося в точке `INV` *lab inv*; имеет в качестве результата:

- $vcs = \{f\}$, где условие корректности f имеет вид:

Implies(
And(*True*, `TACT == 100`, `ON` == *True*,
`OFF` == *False*, `STOP_STATE` == 0,
`ERROR_STATE == 1,`
`Ctrl_Waiting == 2,`
`Ctrl_Drying == 3, timer == 0,`
`Ctrl_state == Ctrl_Waiting,`
`Ctrl_timer == 0, dryer[0] == OFF,`
True),
inv);

- множество *vars* включает следующие типизированные переменные и константы:

```
TACT:int, ON:bool, OFF:bool, val:value,
STOP_STATE:int, ERROR_STATE:int,
Ctrl_Waiting:int, Ctrl_Drying:int,
Ctrl_state:int, Ctrl_timer:int,
dryer:array(bool), hands:array(bool),
timer:int, timer_1:int,
Ctrl_state_1:int, Ctrl_timer_1:int;
```

- *reached* = {*lab*}.

Другие семь условий корректности, начинающиеся в точке INV *lab inv*; и заканчивающиеся в той же самой точке и соответствующие различным ветвям оператора switch и операторов if, порождаются аналогичным образом. Все порожденные условия корректности успешно доказаны в Z3py.

Порождение условий корректности для VOL-программ — результатов трансляции аннотированных Reflex-программ — и их доказательство

завершают описание нашего метода дедуктивной верификации Reflex-программ.

6. ЗАКЛЮЧЕНИЕ

В этой работе мы предложили метод дедуктивной верификации Reflex-программ. Этот метод включает языки аннотаций и аннотаторов для Reflex-программ, ориентированные на описание временных свойств этих программ, алгоритм трансляции аннотированной Reflex-программы в VOL-программу, язык аннотаторов и алгоритм порождения условий корректности для VOL-программ.

Наш метод верификации имеет несколько отличительных свойств. Во-первых, он моделирует взаимодействие между Reflex-программой и управляемым ею объектом через входные и выходные порты, связанные с программными переменными. Инструкция havoc языка VOL позволяет моделировать запись значений из входных портов в программные переменные. Эти внешние значения переменных ограничиваются аннотатором `ASSUME`. Проверка значений, читаемых из программных переменных в выходные порты, специфицируется аннотаторами `ASSERT` и `INVARIANT`. Во-вторых, этот метод сводит верификацию ряда временных свойств Reflex-программ к дедуктивной верификации императивных программ за счет явного моделирования времени в VOL-программах – результатах трансляции Reflex-программ – с помощью глобальных и локальных таймеров, а также историй значений Reflex-переменных, представленных массивами. В-третьих, наш алгоритм порождения условий корректности может применяться к бесконечным циклам, свойственным системам управления.

Имеются несколько направлений для дальнейшего развития метода. Мы планируем расширить его на текстовые языки стандарта IEC 61131-3. Подобно языку Reflex, эти языки используются для программ, которые взаимодействуют с объектом управления только между программными циклами.

Другое направление состоит в исследовании новых временных свойств, для которых верификация также может быть сведена к дедуктивной верификации. Особенно нас интересуют временные аспекты, связанные с историей значений состояний процессов и локальных таймеров процессов, которые позволили бы оценить производительность Reflex-программ.

Явное моделирование времени в Reflex-аннотациях не очень удобный способ описания временных свойств Reflex-программ. Мы планируем использовать временные логики (LTL, CTL and MTL) и их расширения чтобы описывать эти свойства более естественным образом и разработать алгоритм трансляции таких описаний в формулы с явным моделированием времени. Чтобы сделать эту задачу выполнимой, мы планируем использовать специализированные онтологические паттерны [36] вместо произвольных формул этих логик.

В дополнение к решателю Z3 мы намерены использовать другие средства машинной поддержки доказательства чтобы расширить класс проверяемых свойств. В частности, решатель Z3 не смог доказать свойство, что сушилка будет работать в течение не менее 10 секунд после удаления рук, поскольку это свойство требует расширенной индукции. Интерактивный доказатель теорем ACL2 [37] с более развитыми индукционными схемами является хорошим кандидатом для решения этой проблемы.

Чтобы упростить доказательство корректности алгоритмов трансляции и порождения условий корректности, мы планируем использовать унифицированное графо-онтологическое представление для программ на языках Reflex и VOL. Такое представление позволяет свести трансляционную семантику к трансформационной в рамках одного общего графо-онтологического языка и применить онтологический подход для разработки операционной семантики этих языков [38].

7. БЛАГОДАРНОСТИ

Исследование выполнено в рамках темы госзадания ИАиЭ СО РАН (№ АААА-А19-119120290056-0) и при финансовой поддержке РФФИ в рамках научных проектов 17-07-01600 и 20-01-00541.

Acknowledgments: The reported study was funded by State budget of the Russian Federation (IAE project No. АААА-А19-119120290056-0), by RFBR, project number 17-07-01600 and project number 20-01-00541, and by BMBF (Germany) within the project HPC2SE.

СПИСОК ЛИТЕРАТУРЫ

1. *Blanke M., Kinnaert M., Lunze J., Staroswiecki M.* Diagnosis and Fault-Tolerant Control. 2nd edn. Springer-Verlag Berlin, Heidelberg. 2006.
2. IEC 61131-3: Programmable controllers Part 3: Programming languages. Rev. 2.0. Intern. Electrotechnical Commission Std. 2003.
3. *Basile F., Chiacchio P., Gerbasio D.* On the Implementation of Industrial Automation Systems Based on PLC // IEEE Transactions on Automation Science and Engineering, 2013. V. 4. № 10. P. 990–1003.
4. *Thramboulidis K., Frey G.* An MDD Process for IEC 61131-based Industrial Automation Systems // 16th IEEE Intern. Conf. on Emerging Technologies and Factory Automation (ETFA11), Toulouse, France, 2011. P. 1–8.
5. IEC 61499: Function Blocks for Industrial Process Measurement andControl Systems. Parts 1–4. Rev. 1.0. Intern. Electrotechnical Commission Std., 2004–2005.
6. *Wagner F., Schmuki R., Wagner T., Wolstenholme P.* Modeling Software with Finite State Machines. Auerbach Publications. USA, Boston, MA. 2006.

7. *Samek M.* Practical UML statecharts in C/C++: event-driven programming for embedded systems. 2nd edition. 2009. Newnes, Oxford.
8. Control Technology Corporation. QuickBuilder™Reference Guide. 2018. https://controltechnologycorp.com/docs/QuickBuilder_Ref.pdf. Last accessed 20 Jan 2019
9. *Zyubin V.E.* Hyper-automaton: A Model of Control Algorithms // Proceedings of the IEEE Intern. Siberian Conf. on Control and Communications (SIBCON-2007). The Tomsk IEEE Chapter & Student Branch. Tomsk, Russia, 2007. P. 51–57.
10. *Hoare C.A.R.* Communicating Sequential Processes. Prentice-Hall Int., 1985.
11. *Harel D.* Statecharts: a Visual Formalism for Complex Systems // Science of Computer Programming, 1987. V. 8. № 3. P. 231–274.
12. *Lynch N., Tuttle M.* An Introduction to Input/Output Automata // CWI Quarterly, 1989. V. 2. № 3. P. 219–246.
13. *Berry G.* The Foundations of Esterel // Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press, Foundations of Computing Series, 2000. P. 425–454.
14. *Henzinger T.A.* The Theory of Hybrid Automata // Inan M.K., Kurshan R.P. (eds) Verification of Digital and Hybrid Systems, NATO ASI Series (Series F: Computer and Systems Sciences), Springer, Berlin, Heidelberg, 2000. V. 170. P. 265–292.
15. *Milner R.* Communication and Concurrency. Series in Computer Science. Prentice Hall, New Jersey. 1989.
16. *Kaynar D.K., Lynch N., Segala R., Vaandrager F.* Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems // 24th IEEE Intern. Real-Time Systems Symposium (RTSS'03), 2003, IEEE Computer Society Cancun, Mexico. P. 166–177.
17. *Kof L., Schätz B.*: Combining Aspects of Reactive Systems // Proc. of Andrei Ershov Fifth Int. Conf. Perspectives of System Informatics, Novosibirsk, 2003. P. 239–243.
18. *Zyubin V.* SPARM Language as a Means for Programming // Microcontrollers. Optoelectronics, Instrumentation, and Data Processing, 1996. V. 2. № 7. P. 36–44.
19. *Liakh T.V., Rozov A.S., Zyubin V.E.* Reflex Language: a Practical Notation for Cyber-Physical Systems // System Informatics, 2018. V. 12. № 6. P. 85–104.
20. *Rozov A.S., Zyubin V.E.* Process-oriented programming language for MCU-based automation // Proc. of the IEEE Intern. Siberian Conf. on Control and Communications, The Tomsk IEEE Chapter Student Branch, Tomsk, Russia, 2013. P. 1–4.
21. *Bulavskij D., Zyubin V., Karlson N., Krivoruchko V., Mironov V.* An Automated Control System for a Silicon Single-Crystal Growth Furnace // Optoelectronics, instrumentation, and data processing, 1996. V. 2. № 5. P. 25–30.
22. *Travis J., Kring J.* LabVIEW for Everyone: Graphical Programming Made Easy and Fun. 3rd Edition. Prentice Hall PTR, Upper Saddle River, NJ, USA. 2006.
23. *Zyubin V.* Using Process-Oriented Programming in LabVIEW // In: Proc. of the Second IASTED Intern. Multi-Conference on "Automation, control, and information technology": Control, Diagnostics, and Automation.Novosibirsk, 2010. P. 35–41.
24. *Randell B.* Software Engineering Techniques // Report on a conference sponsored by the NATO Science Committee. Brussels, Scientific Affairs Division, NATO, Rome, Italy, 1970. P. 16.
25. Z3 API in Python, https://ericpony.github.io/z3py-tutorial/guide-examples.htm Last accessed 20 Jan 2019
26. *Moura L., Bjurner N.*: Z3: An Efficient SMT Solver. TACAS 2008: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, 2008. V. 4963. P. 337–340.
27. *Anureev I.S., Garanina N.O., Liakh T.V., Rozov A.S., Zyubin V.E., Gorlatch S.* Two-Step Deductive Verification of Control Software Using Reflex // Proceedings of A.P. Ershov Informatics Conference (PSI-19). A.P. Ershov Institute of Informatics Systems: IPC NSU, Novosibirsk, Russia, LNCS, 2019. V. 11964. P. 50–63.
28. *Barnett M., Chang B.-Y.E., DeLine R., Jacobs B., Leino K.R.M.* Boogie: A Modular Reusable Verifier for Object-Oriented Programs // In: Proc. of the 4th Intern. Conf. on Formal Methods for Components and Objects, LNCS. 2005. V. 4111. P. 364–387.
29. FramaC Homepage, https://frama-c.com/
30. Spark Pro Homepage, https://www.adacore.com/sparkpro
31. The KeY project Homepage, https://www.key-project.org/
32. Dafny Homepage, https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/.
33. *Dijkstra E.W., Scholten C.S.* Predicate Calculus and Program Semantics, Springer-Verlag, 1990.
34. *Nepomniaschy V., Anureev I., Mikhailov I., Promsky A.* Towards verification of C programs. C-light language and its formal semantics // Programming and Computer Science, 2002. V. 28. № 6. P. 314–323.
35. *Nepomniaschy V., Anureev I., Promsky A.* Towards verification of C programs: Axiomatic semantics of the C-kernel language // Programming and Computer Science, 2003. V. 29. № 6. P. 338–350.
36. *Garanina N., Zyubin V., Lyakh V., Gorlatch S.* An Ontology of Specification Patterns for Verification of Concurrent Systems // In: New Trends in Intelligent Software Methodologies, Tools and Techniques // Proceedings of the 17th International Conference SoMeT-18, Series: Frontiers in Artificial Intelligence and Applications, Amsterdam: IOS Press, 2018. P. 515–528.
37. ACL2 Homepage, http://www.cs.utexas.edu/users/moore/acl2/
38. *Anureev I.S.* Operational ontological approach to formal programming language specification // Programming and Computer Software, 2009. V. 35. № 1. P. 35–42.